Exploring the Link Between Test Suite Quality and Automatic Specification Inference*

Luke Chircop
CS, ICT, University of Malta
luke.chircop@um.edu.mt

Christian Colombo
CS, ICT, University of Malta
christian.colombo@um.edu.mt

Mark Micallef
CS, ICT, University of Malta
mark.micallef@um.edu.mt

While no one doubts the importance of correct and complete specifications, many industrial systems still do not have formal specifications written out — and even when they do, it is hard to check their correctness and completeness. This work explores the possibility of using an invariant extraction tool such as Daikon to automatically infer specifications from available test suites with the idea of aiding software engineers to improve the specifications by having another version to compare to. Given that our initial experiments did not produce satisfactory results, in this paper we explore which test suite attributes influence the quality of the inferred specification. Following further study, we found that instruction, branch and method coverage are correlated to high recall values, reaching up to 97.93%.

1 Introduction

Having formally written specifications that are correct and complete is important. However, many industrial systems still do not have formal specifications written out. Even when they do, due to the size and complexity of such industrial systems, checking for correctness and completeness of the specification is rare.

Obtaining formally written specifications can be beneficial for multiple purposes. These can be used by a developer or tester as a point of reference to verify the implementation's behaviour. Formal specifications can also be exploited to aid test case generation [25] by pinpointing deficiencies in the automatically generated tests, mutation testing [23] by providing the means to identify valuable mutations minimising equivalent mutants, regression testing [16] by pinpointing unintentional behavioural differences between different versions of the same system and to track down software bugs [1, 13, 21].

Instead of expecting companies or forcing developers to write formal specifications, there has been extensive research carried out [3, 5, 6, 9, 17, 19, 22] for the possibility of automatically inferring specifications from readily available information. Apart from having to learn how to use the tool and prune some of the inferred specifications, acquiring the ability to infer specifications with minimal effort may encourage the industry to exploit the technology and use it to their advantage.

Specifications can be inferred by using one or a combination of static and dynamic inference techniques. Static inference techniques process the source code and infer specifications based on the observations that are made. Dynamic inference techniques on the other hand, make use of drivers to exercise the system and specifications are inferred based on how the system is observed to react. A popular driver choice for such inference techniques are test suites.

Given that around half of the development time for industrial systems is spent on testing, multiple test suites would have been written and be readily available. Furthermore, such test suites are purpose built to exercise as much of the system as possible, taking into consideration multiple input/output combinations

*Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013

Adrian Francalanza, Gordon J. Pace (Eds.): Second International Workshop
on Pre- and Post-Deployment Verification Techniques (PrePost 2017)
EPTCS 254, 2017, pp. 44–56, doi:10.4204/EPTCS.254.4

to expose bugs in the system. This makes them an ideal source to use as drivers and gather knowledge with regards to how the system is expected to behave. Moreover, if used after all bugs identified (if any) are fixed, it can be safe to assume that the behaviour being observed is correct and will reduce the possibility of inferring incorrect specifications.

One popular dynamic inference tool that has been used for various applications [3, 16, 18, 19] is Daikon [9]. Daikon, is an invariant inference tool that makes use of test suites or other suitable resources to exercise systems and infer specifications based on what it observes. Since dynamic inference techniques infer program properties based on what is observed, the quality of the test suite greatly impacts the resulting correctness of the inferred specification. In fact, various testing attributes (used to measure the quality and effectiveness of written tests) including only testing for correct behaviour or covering all branch conditions could potentially have an effect on the outcome.

In this paper, we make use of the Daikon tool to explore which test suite attributes influence the quality of the inferred specification. By doing so, this work will be able to provide clear and concise guidelines which may prove useful to users who want to infer and make use of specifications that correctly represent how a system should behave, by guiding them to focus on test suite attributes that were found to have the highest impact on specification quality. Moreover, such correlations can be used to predict the level of quality and correctness of the resulting specifications. Line and branch coverage are being hypothesised as potential test attributes that could have a noticeable affect on the quality of inferred specifications.

In the rest of the paper, we present the state of the art in specification inference techniques and explore links between specification inference and test suite quality. More specifically, we start by introducing various approaches that exist to tackle this problem and discuss the reasoning behind the choice of the technique considered for the initial experiments in Section 2. Subsequently, we introduce the case study (Section 3) used throughout the evaluation process and report the empirical results in Section 4. Finally, we discuss related work (Section 5), future work (Section 6.1) and conclude the paper (Section 6).

2 Inferring Program Specifications

The ultimate goal for any automatic specification inference technique is to be able to infer both complete and correct specifications on their own without any human intervention. However, it is both intractable and NP-hard as proven by Gold [11] to have a program that determines on its own whether properties inferred, over or under generalises correct behaviour.

Nonetheless, many techniques and tools have been introduced [9, 13, 16, 22] that automatically approximate program specifications based on information that is provided or observed. Two varying approaches divide the state of the art: static and dynamic inference techniques.

2.1 Static inference

Static inference techniques infer program properties by examining the software's source code and understanding how variables and methods interact with each other. Ramanathan et al. [22], for example, introduced data flow analysis to infer control-flow and data-flow specifications able to represent the order in which methods can be executed and the state that variables have to be in upon or after execution of a method. Other works [4, 10], make use of static inference techniques to discover common programming errors and validate program properties.

This approach can be inefficient and may also require explicit goals or special annotations to be

included in the source code; making it difficult and tedious at times [17]. Furthermore, the technique is not able to reason about behaviour that is runtime dependent.

2.2 Dynamic inference

On the other hand, dynamic inference techniques [3, 6, 9, 16, 17] infer program specifications by exercising the software and observing how it reacts to various inputs. Numerous sources can be used to exercise the software, however, test suites are most commonly used due to the implicit knowledge they hold.

During the first phase of the approach, the software is exercised using the selected source driver and a trace is recorded. The recorded trace represents the program and data-flow of the exercised software. These traces are then processed in the second phase of the approach, whereby generalised specifications are inferred based on the behaviour that was observed. Program specifications have been modelled in multiple ways including: finite state machines using techniques such a k-tail [2] and invariants representing the manner in which methods can interact with each other and defining the state which variables should be in before or after the execution of methods, respectively.

An assortment of tools exist that implement the dynamic inference approach including the Agitator tool [3] that infers invariants using guided test input generation in an iterative process, the Behaviour, Capture and Test tool [16] that exercises software and exploits the invariants generated to aid bug finding during regression testing and iDiscovery [26] which combines dynamic inference and symbolic execution in an iterative feedback process to continuously refine the inferred specification until a fixed point is reached. All tools have one thing in common, that is, all make use of a tool called Daikon [9].

Daikon was first introduced by Michael D. Ernst et al. in 2000. It is able to infer invariants by instrumenting the software, exercising the instrumented software (e.g., using test suites), and finally inferring specifications based on the recorded traces. This was achieved by having the inference phase test the recorded values of instrumented variables against a predefined set of possible invariants [9, 17]. Furthermore, some pruning and statistical techniques are implemented to reduce the number of undesirable invariants and present potentially relevant ones [8].

As stated in [17] and many other papers [9, 20, 26], the accuracy and correctness of the specifications that are inferred by dynamic inference tools are in part dependent on the quality and effectiveness of the test suites used to exercise the software during the trace recording process. This paper makes use of the Daikon tool to explore which test suite attributes have a positive influence on the quality of the inferred specifications.

3 Case Study

To be able to discover correlations between test suite attributes and their effect on the quality of inferred specifications, a financial transaction system (FTS in short) [1] was designed and built. Included in the FTS case study is: (i) the software containing core functionality, (ii) a combination of four test suites with varying attributes testing the functionality of the software, and (iii) manually written specifications representing the desired behaviour of the software.

Software: The core functionality of the FTS software was designed and built to replicate how a system would have been built in industry containing 1549 lines of code. Therefore, a multi level system

[1] The implementation is available at: https://github.com/lukechircop007/AutomaticSpecificationInferenceEvaluation.git

with a facade implementation approach was taken. The API functionality accepts requests for: new user profiles to be created, accounts to be opened or closed and monetary transactions to and from local or foreign accounts to be performed, whilst also providing different user modes and account types that impact the manner in which some requests and charges are serviced. Moreover, the core functionality was also built to correctly handle invalid inputs or requests by throwing exceptions and returning pre-defined values, amongst others abilities. This was designed to showcase whether specification inference tools are able to infer specifications describing the handling of invalid requests or input.

Test Suites: Test suites were also built alongside the core functionality mimicking how the software would have been tested in industrial companies. To this end, unit tests (containing 437 lines of code) were introduced to separately validate the behaviour of low-level software functionality. Subsequently, integration tests (containing 1648 lines of code) were also introduced using a top-down approach to validate the high-level functionality that combined and interacted with various parts of the software. Test-to-pass (validating good behaviour) and test-to-fail (validating correct handling of invalid input or requests) tests were introduced at each level (marked as Valid and Invalid) to increase the total overall coverage of the software's behaviour. Boundary based testing techniques were also used whenever applicable to ensure coverage of corner cases.

Every test was kept as simple as possible only exercising and checking for one property at a time with supporting set-up and tear-down functions. Furthermore, the inputs where pre-defined and fixed, facilitating repeatability of test runs (if bugs were to be uncovered).

Specifications: Finally, a specification was manually written representing how the software's functionality should behave given any input. This, provided the means to measure how much Daikon and other inference tools are able to infer correct and complete specifications by comparing the inferred specification with the manually written one and identifying matching, non matching and incorrect conditions. For this to be achieved, conditions of what input to accept and reactions to observe were defined for every method in the software. Some examples of such conditions include expecting an *amount* value that is greater or equal to zero and expecting the *balance* value to increase by *amount* value after a successful *deposit* request. The complete specification is provided within the FTS repository.

4 Discovering Correlations between Specifications and Test Suites

A selection of distinguishable test suite attributes and the resulting correctness of inferred specifications had to first be selected and evaluated in order to discover correlations between test suites and quality of specification. After obtaining the results, the study to discover correlations between test suites and specifications was conducted. Throughout the experiment, a developer machine consisting of an i7 quad core processor, eight gigabytes of RAM, running Windows 10 and the latest versions of Java (Java 8) was used.

The rest of the section, discusses the design decisions involved (Sections 4.1 and 4.2). Subsequently, we present and interpret the results and correlations that were identified in Sections 4.2.1, 4.2.2 and 4.2.3.

4.1 Choosing Relevant Test Suite Attributes

Test suite attributes play an important role in the discovery of correlations since they are used to measure the quality and effectiveness of test suites allowing for one to be distinguishable from the other. A wide

selection of metrics already exist that are able to showcase test suite attributes and have been used to evaluate how effective the written tests are at testing the software they were aimed for. Some metrics include measuring the percentage of software instructions exercised by the test suite and the percentage of conditional branch combinations covered.

Rather than re-inventing the wheel, the same metrics were considered for the study. Further to the metrics mentioned above, line and method coverage were also taken into consideration. Since dynamic inference techniques infer specifications by analysing software behaviour, it stands to reason that having high software coverage levels will increase the possibility of discovering specifications that are of greater quality and completeness. In fact, we were hypothesising that having high branch and line coverage can positively influence the quality and correctness of the inferred specifications since the daikon tool is non-monotonic allowing conditions to be retracted based on further observed evidence.

4.2 Running the Experiment

To determine the influence that test suite attributes had on the quality of inferred specifications, multiple test suites with varying metric values (some low, some high) were introduced resulting in a total combination of nine runs.

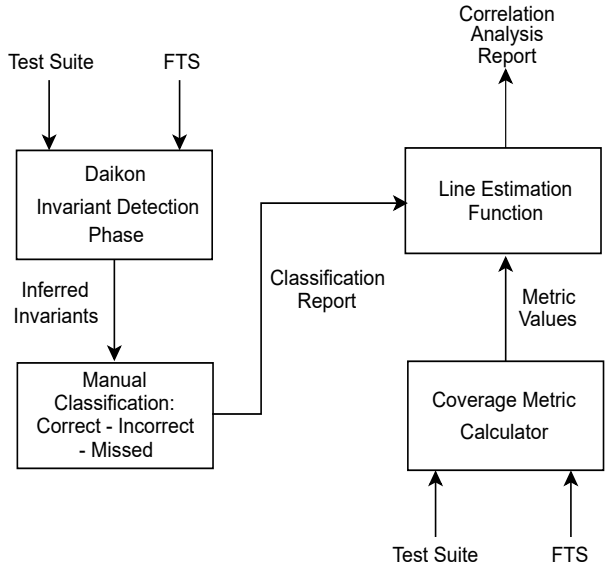

Figure 1: Evaluation process carried out to gather results for the discovery of potential correlations between test suite attributes and quality of inferred specification

As shown in Figure 1, for each run:

- Test suite metrics discussed in Section 4.1 were measured.
- Subsequently, Daikon was instructed to instrument a fresh version of the FTS and use the unique combination of tests defined for the particular run to infer a set of invariants.
- Once inferred, the invariants were classified as correct or incorrect and a list of missing invariants was also included.

The results were then used to discover which test suite attributes positively influenced the quality of inferred specifications.

4.2.1 Coverage Metrics

As mentioned in Section 3, test-to-pass (marked as Valid) and test-to-fail (marked as Invalid) tests were introduced in both unit and integration tests. For the purpose of this study, a combination of valid, invalid, unit and integration tests was introduced for each run. When compared, the varying runs showcase distinctive coverage metric values with the last run encapsulating all tests, having the best coverage, as shown in Table 1.

Test Suites Executed	Total Tests	Instruction (%)	Branch (%)	Line (%)	Method (%)
Valid: UT	37	22.20	11.40	20.80	46.50
Invalid: UT	9	14.40	7.20	12.00	20.80
Valid & Invalid: UT	46	23.20	16.90	21.10	47.50
Valid: IT	50	70.00	61.40	80.10	85.10
Invalid: IT	44	76.80	67.50	77.20	77.20
Valid & Invalid: IT	94	90.10	86.70	94.90	91.10
Valid: UT & IT	87	70.80	62.00	80.90	88.10
Invalid: UT & IT	53	78.30	71.10	78.30	81.20
Valid & Invalid: UT & IT	140	91.20	90.40	95.70	94.10

Table 1: Coverage metrics measured for varying runs throughout the experiment

The first column displays the combination of unit (UT), integration (IT), valid and invalid request handling tests considered for each run. The second column presents the number of tests executed for each run. Finally, the remaining columns represent various coverage metrics including: instruction, branch, line and method coverage.

4.2.2 Evaluating Correctness of inferred Specifications

Deciding whether the inferred invariants for each run were correct or incorrect required some thought.

Correct invariants: Consider a `deposit` method that accepts an *amount* value to be added to an existing *balance*, as shown in Listing 1. A property that could be defined for such a function would be that the `amount` variable upon execution of the *deposit* method has to be greater or equal to zero.

```
1  public class UserAccount {
2     protected boolean opened;
3     protected String account_number;
4     protected double balance;
5     ...
6
7     public void deposit(double amount)
8     {
9        balance += amount;
10    }
11    ...
```

Listing 1: Snippet code of a deposit function

```
1  public class UserAccountUTPass {
2
3     Integer userId = 03;
```

```
4     String accountNumber = "45678";
5     double startingBalance = 35.0;
6     double amountToDeposit = 100.0;
7     ...
8
9
10    @Test
11    public void deposit_Test_Pass(){
12       userAccount.balance = startingBalance;
13       userAccount.deposit(amountToDeposit);
14       assertTrue(userAccount.balance == (startingBalance + amountToDeposit));
15    }
16    ...
```

Listing 2: Snippet code of a deposit test function

An invariant inferred by the tool when exercising the software using the test shown in Listing 2 that would comply with the latter property, would be:

$$\text{amount} >= 35.0$$

However, as one would notice, the inferred invariant does not fully represent the set of all acceptable input since it would not accept values between zero and thirty four. Such an invariant is deemed to over approximate, limiting the range of acceptable input. Similarly, there can be under approximating invariants that include the set of acceptable input whilst also accepting some undesirable ones [24]. Since the study wanted to determine to what extent the inference technology was able to infer complete specifications, conditions were therefore classified as correct only if they did not over or under approximate at least one property inside the manually written specification.

Throughout the evaluation process however, some inferred conditions were encountered that correctly expressed how the system should behave but did not coincide with any property from the manually written specification. In such instances, the observed conditions were deemed to be correct and added to the manually written specification, expecting the other runs to also infer the newly added condition.

Incorrect Invariants: The remaining conditions inside the inferred specifications were then classified as incorrect.

Missing Invariants: Having classified all the inferred conditions, the properties residing inside the manually written specification that did not have a matching invariant were marked as missed.

The resulting classifications for each run can be found in Table 2.

						Original Spec		Original + Extra Spec	
Runs	Correct (Orig)	Correct	Incorrect	Missed(Orig)	Missed	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Valid: UT	448	473	982	469	755	31.32	48.85	32.51	38.52
Invalid: UT	188	199	517	729	1029	26.67	20.50	27.80	16.21
Valid & Invalid: UT	481	507	1032	436	721	31.79	52.45	32.94	41.29
Valid: IT	854	903	2028	63	325	29.63	93.13	30.81	73.53
Invalid: IT	604	658	1627	313	570	27.07	65.87	28.80	53.58
Valid & Invalid: IT	878	934	2294	39	294	27.68	95.75	28.93	76.06
Valid: UT & IT	863	912	2011	54	316	30.03	94.11	31.20	74.27
Invalid: UT & IT	673	727	1829	244	501	26.90	73.39	28.44	59.20
Valid & Invalid: UT & IT	898	954	2340	19	274	27.73	97.93	28.96	77.69

Table 2: Classification results for multiple runs including precision and recall calculations

The first column shows the combination of unit, integration, valid and invalid request testing that were considered for each run. The second column represents the number of inferred invariants observed to

correctly represent properties defined in the manually written specification, whilst the fifth column, shows the number of properties defined in the specification that were not represented. The greyed third, fourth and sixth columns, present the total number of correct (including the extra invariants added to the original expected set), incorrect and missed (including those missed from the extra invariants) invariants that were observed. Finally, the last four columns present the precision and recall of the inferred invariants based on the original reported findings and the combination of original with the extra invariants(grey columns) respectively.

The difference between the varying runs can be clearly seen by the number of correct versus missing conditions presented. Runs that only considered unit tests faired the worst, observing a higher number of missing conditions than correct ones. On the other hand, runs that made use of integration tests performed better, some even providing over 900 correct conditions. The last run that combined all unit and integration tests faired the best, exhibiting the highest number of correct conditions observed at 954 versus 274 missed for the combined results and 898 correct versus 19 missed for the original spec comparison. The recall column illustrates this, registering an increase in quality of over 77.43 and 61.48 percentage points for the original and combined results over the worst performance of 20.50 and 16.2 percent at 97.93 and 77.69 percent respectively. However, the precision column did not present similar increases. Instead, the precision percentage was observed to decrease as the complexity of the runs increased. This occurred since precision takes into consideration the number of incorrect invariants, whom kept increasing as the coverage values increased. Moreover, recall percentages were observed to be higher when taking into account the classification reports that do not include the extra invariants.

4.2.3 Discovering test suite attributes that influence quality of inferred specification

Once all the metrics for each run were measured and invariants classified, the study to discover correlations between high quality specification inference and test suite attributes was carried out.

The precision and recall values along with the coverage metrics presented in Tables 2 and 1 were passed on to a linear estimation function to identify correlations between the two. As expected, instruction, method and branch coverage attributes were found to have a high correlation to the quality of the inferred specification with an 0.8956 determination coefficient for recall. This was as (discussed in Section 4.1) expected since such high coverage values indicate that a greater chunk of the software was exercised, increasing the behaviour observed and chances of inferring useful invariants

The linear estimation function found a clear correlation, however, having high instruction, method and branch coverage does not ensure that the inference tool will produce the optimal set of invariants. In fact, although the number of correct conditions observed (for the optimal run) was considerable, many of the correct invariants did not represent the manually written specifications. Instead, most consisted of additional invariants that had been inferred by Daikon and added throughout the classification process since they were found to be correct and useful. Furthermore, it was observed that some properties defining major functionality of the software were never observed, highlighting a number of deficiencies and limitations.

One limitation that was observed whilst identifying the missed conditions, was the ability to reason about complex data types such as *ENUMS* and array lists of complex data types. Hence, a sizeable amount of the specification was never met. Moreover, the Daikon tool was observed to ignore the inference of invalid input handling whenever exceptions were thrown by the system.

Other complex properties such as

Return true only if the User ID is valid, Session ID is open, account_number exists, status of account is set to open, the amount is greater than zero and there are sufficient funds for a withdrawal request

were not observed. However, this was expected since such properties involve the combination of more than three variables for one invariant, which is one major limitation that the Daikon tool has. Moreover, for a tool to infer such a property, some level of knowledge with regards to what the data represents would be required which is hard to obtain without requiring human aid.

Throughout the classification process, there were also some conditions which were expected but never observed including:

$$\text{orig(balance)} + \text{amount} == \text{balance}$$
$$\text{uid} >= 0$$
$$\text{amount} >= 0$$

Having a property checking that the balance is equal to the original balance plus the amount, or checking that the user id (*uid*) and *amount* is greater than zero are all examples of invariants that the Daikon tool is known to be capable of inferring. However, they did not appear in any of the nine runs evaluated. Upon further investigation, it was discovered that the Daikon tool requires a more diverse test set for such properties to be inferred. This was confirmed since incorrect conditions like "amount one of {200,100,50}" were observed confirming that the tool was not able to further generalise the invariant and required more diverse values to do so.

Therefore, having high instruction, method and branch coverage does not ensure that the inference tool will produce the optimal set of specifications. Nonetheless, they contribute to the quality of inferred specifications.

4.3 Discussion: Lessons learned

We have discovered through the tool agnostic experiment presented in this paper that instruction, method and branch coverage metrics influence the quality of specifications inferred. However, they do not ensure that the specifications inferred will be correct.

Given the observations that we made, using fixed values to test similar behaviour is not recommended. Instead, a wider input selection should be used to increase the chances of inferring better specifications. However, due to the inference nature of tools such as Daikon, using a wider range of input may not be enough. The additional inputs distributed across tests validating different properties, would still provide limited samples ending up with inaccurate invariants. Therefore, in addition to using a wider rage of input, we would also recommend to increase the number of times the software's functionality is exercised using unique inputs each time. Hence providing more samples for the same behaviour.

Taking an equivalent partitioning approach and running the test suite multiple times may also increase the chances of inferring better quality specifications since a potentially wide selection of inputs for each corner case and branch opportunity would be available providing multiple unique samples from which specifications can be correctly inferred.

By providing more samples, tools such as Daikon, will be able to refine the invariant generalisation process and potentially increase the number of correct invariants. Moreover, the increase in unique samples may also drastically reduce the number of incorrect invariants since the chances of having an incorrect invariant invalidated by a sample during the inference process also increases.

5 Related Work

Several studies have been conducted to determine whether test suite attributes influence the quality of specifications inferred. Harder, Mellen and Ernst [14], presented a new metric that can be used to write,

generate and minimise test suites. To evaluate this new approach, various existing metrics including statement and branch coverage were taken into consideration and compared. The comparison was carried out by instructing Daikon to infer invariants for faulty and correct versions of C programs driven by test suites written to obtain high respective coverage values. Differences in effectiveness were obtained by "measuring the proportion of times the fault was detected" by [14] each test suite. For our purpose, the quality of invariants inferred using varying test suites cannot be measured by counting the number of faults detected. This is due to the fact that our end goal is that to obtain a specification. Therefore, it is of the utmost importance that the system operates as intended without any faults to be able to correctly infer the specification. Nonetheless, the same conclusions were drawn. Having 100% statement and branch coverage was confirmed to aid fault finding. However, if other metrics such as the one introduced by their study is combined, further improvements can be observed. This conforms with the observations made by this study that using other metrics may potentially refine and produce more accurate invariants.

Similarly, Gupta and Heidepriem [12], introduced the *invariant-coverage* criterion aimed to improve the accuracy of dynamically inferred specifications. The authors made use of five programs covering a variety of program features such as calculating the Levenshtein distance between two strings and calculating the adjoint of a matrix. Throughout the evaluation process, the specifications inferred when using traditional branch and definition-use pair suites were compared to the outcome of the newly introduced criterion. Although only traditional coverage metrics were taken into consideration for our study, a larger program was used to challenge both the abilities of the specification inference tool and the test suites used to infer the specifications. Moreover, a different approach was taken to compare the effectiveness of using different coverage metrics. Instead of comparing the inferred invariants to a formally written specification, the authors identified meaningful invariants that are not "linked to specific input values used and are abstractions over multiple inputs". Nevertheless, Gupta et al. showed that *invariant-coverage* suites were able to infer higher quality specifications eliminating various false invariants inferred when using traditional criteria. Confirming yet again the results of the study that we carried out.

Polikarpova, Ciupa and Meyer [20], performed experiments to compare properties that are written by developers and testers versus those inferred automatically by tools such as Daikon. Much like the experiment that was carried out by this paper, the authors measured the proportion of user written properties implied by the inferred specification and the number of incorrect invariants inferred, finding correlations between code metrics and the quality of contracts inferred. The total recall was that of 59% for the large test suite whilst negative correlations were observed between correctness and coupling. Positive correlations on the other hand were observed between assertion clauses inferred and the number of relevant inferred invariants.

6 Conclusion

No one doubts the importance of formally written specifications. However, such specifications are rarely written out and when they are, checking their correctness and completeness is hard. Instead of forcing developers or testers to write such specifications, various tools including Daikon have explored the possibility of automatically inferring generalised specifications based on sources of information that are provided including test suites. Given the nature of such sources, the pool of information may be limited, which in turn reflects negatively upon the outcome of the specifications inferred.

This study carried out an experiment to discover test suite attributes that positively influence the quality of the inferred specifications. Results indicated that there was a strong correlation between having high line, method and branch coverage and high recall values (up to 97.93 %). However, even though

the quality of the inferred specifications was high, it was also observed that having high coverage values does not ensure that optimal specifications will be inferred. As a consequence, incorrect and missing invariants were still observed highlighting the importance of having a varied selection of inputs from which such techniques can correctly generalise the expected behavioural conditions. In fact, this study can be applied to other specification inference tools whereby the same or similar results are expected.

6.1 Future Work

This paper presents precision and recall values for specifications inferred by Daikon when provided with varying test suites. As discussed in Section 4.2.3, in addition to having high instruction and branch coverage, a wider selection of inputs and exercises are required for Daikon to be able to successfully infer correct specifications. In future work, we aim to investigate other possible approaches that can be used to automatically introduce such inputs and drive through the use of techniques such as random and concolic testing aming to obtain the optimal specification. Metrics presented by Gupta et al. [12] and Harder et al. [14] could also be used alongside such techniques to enhance the quality of specifications inferred.

Furthermore, after concluding the experiments on the small case study, we would like to carry out a similar test on a larger industrial system to evaluate whether the industry is ready to adopt the technology and provide guidelines on how the test suites should be written to ensure optimal specification inference results.

Iterative feedback loops [26] are also of interest to us since the existing test suites could potentially be used as the initial "seeds" from which the process can start inferring a specification and gradually refine it every iteration. Moreover, we are not excluding the possibility of exploring other techniques that do not use Daikon such a DySy [7] which combines concrete executions of test suites with symbolic execution to abstract conditions over program variables satisfied by tests during execution.

Finally, we would also like to explore other possible sources of information that could be introduced to the trace recording phase providing a richer understanding of how the system behaves. An example of sources that we are considering include live system runs or execution logs. By taking such sources into consideration, we are hypothesising the possibility that the number of incorrect invariants classified would decrease drastically since many of the incorrect invariants would be invalidated by traces that contradict them. In addition, some of the partially correct invariants could also potentially mutate into correct invariants given the broader set of behaviour observed.

References

[1] Rui Abreu, Alberto González, Peter Zoeteweij & Arjan J. C. van Gemund (2008): *Automatic Software Fault Localization Using Generic Program Invariants*. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, ACM, New York, NY, USA, pp. 712–717, doi:10.1145/1363686.1363855.

[2] A. W. Biermann & J. A. Feldman (1972): *On the Synthesis of Finite-State Machines from Samples of Their Behavior*. *IEEE Transactions on Computers* C-21(6), pp. 592–597, doi:10.1109/TC.1972.5009015.

[3] Marat Boshernitsan, Roongko Doong & Alberto Savoia (2006): *From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing*. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, ACM, New York, NY, USA, pp. 169–180, doi:10.1145/1146238.1146258.

[4] Cristiano Calcagno & Dino Distefano (2011): *Infer: An Automatic Program Verifier for Memory Safety of C Programs*. In: *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, Springer-Verlag, Berlin, Heidelberg, pp. 459–465, doi:10.1007/978-3-642-20398-5_33.

[5] L. Cardamone, A. Mocci & C. Ghezzi (2011): *Dynamic synthesis of program invariants using genetic programming*. In: *2011 IEEE Congress of Evolutionary Computation (CEC)*, pp. 624–631, doi:10.1109/CEC.2011.5949677.

[6] Jake Cobb, James A. Jones, Gregory M. Kapfhammer & Mary Jean Harrold (2011): *Dynamic Invariant Detection for Relational Databases*. In: *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA '11, ACM, New York, NY, USA, pp. 12–17, doi:10.1145/2002951.2002955.

[7] Christoph Csallner, Nikolai Tillmann & Yannis Smaragdakis (2008): *DySy: Dynamic Symbolic Execution for Invariant Inference*. In: *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM, New York, NY, USA, pp. 281–290, doi:10.1145/1368088.1368127.

[8] Michael D. Ernst, Adam Czeisler, William G. Griswold & David Notkin (2000): *Quickly Detecting Relevant Program Invariants*. In: *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, ACM, New York, NY, USA, pp. 449–458, doi:10.1145/337180.337240.

[9] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz & Chen Xiao (2007): *The Daikon System for Dynamic Detection of Likely Invariants*. *Sci. Comput. Program.* 69(1-3), pp. 35–45, doi:10.1016/j.scico.2007.01.015.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe & Raymie Stata (2002): *Extended Static Checking for Java*. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, ACM, New York, NY, USA, pp. 234–245, doi:10.1145/512529.512558.

[11] E Mark Gold (1978): *Complexity of automaton identification from given data*. *Information and Control* 37(3), pp. 302 – 320, doi:10.1016/S0019-9958(78)90562-4. Available at `http://www.sciencedirect.com/science/article/pii/S0019995878905624`.

[12] N. Gupta & Z. V. Heidepriem (2003): *A new structural coverage criterion for dynamic detection of program invariants*. In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 49–58, doi:10.1109/ASE.2003.1240294.

[13] Sudheendra Hangal & Monica S. Lam (2002): *Tracking Down Software Bugs Using Automatic Anomaly Detection*. In: *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, ACM, New York, NY, USA, pp. 291–301, doi:10.1145/581339.581377.

[14] Michael Harder, Jeff Mellen & Michael D. Ernst (2003): *Improving Test Suites via Operational Abstraction*. In: *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, IEEE Computer Society, Washington, DC, USA, pp. 60–71. Available at `http://dl.acm.org/citation.cfm?id=776816.776824`.

[15] Davide Lorenzoli, Leonardo Mariani & Mauro Pezzè (2008): *Automatic Generation of Software Behavioral Models*. In: *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM, New York, NY, USA, pp. 501–510, doi:10.1145/1368088.1368157.

[16] L. Mariani (2004): *Behavior capture and test for verifying evolving component-based systems*. In: *Proceedings. 26th International Conference on Software Engineering*, pp. 78–80, doi:10.1109/ICSE.2004.1317429.

[17] Jeremy W. Nimmer & Michael D. Ernst (2002): *Automatic Generation of Program Specifications*. In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, ACM, New York, NY, USA, pp. 229–239, doi:10.1145/566172.566213.

[18] M. Papadakis & N. Malevris (2010): *Automatic Mutation Test Case Generation via Dynamic Symbolic Execution*. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 121–130, doi:10.1109/ISSRE.2010.38.

[19] Jeff H. Perkins & Michael D. Ernst (2004): *Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants*. In: *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on*

Foundations of Software Engineering, SIGSOFT '04/FSE-12, ACM, New York, NY, USA, pp. 23–32, doi:10.1145/1029894.1029901.

[20] Nadia Polikarpova, Ilinca Ciupa & Bertrand Meyer (2009): *A Comparative Study of Programmer-written and Automatically Inferred Contracts*. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, ACM, New York, NY, USA, pp. 93–104, doi:10.1145/1572272.1572284.

[21] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi & Steven P. Reiss (2003): *Automated Fault Localization Using Potential Invariants*. *CoRR* cs.SE/0310040. Available at `http://arxiv.org/abs/cs.SE/0310040`.

[22] Murali Krishna Ramanathan, Ananth Grama & Suresh Jagannathan (2007): *Static Specification Inference Using Predicate Mining*. *SIGPLAN Not.* 42(6), pp. 123–134, doi:10.1145/1273442.1250749.

[23] David Schuler, Valentin Dallmeier & Andreas Zeller (2009): *Efficient Mutation Testing by Checking Invariant Violations*. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, ACM, New York, NY, USA, pp. 69–80, doi:10.1145/1572272.1572282.

[24] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia & Mark Harman (2012): *Finding the Optimal Balance Between Over and Under Approximation of Models Inferred from Execution Logs*. In: *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, IEEE Computer Society, Washington, DC, USA, pp. 21–30, doi:10.1109/ICST.2012.82.

[25] Tao Xie & David Notkin (2004): *Mutually Enhancing Test Generation and Specification Inference*, pp. 60–69. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-24617-6_5.

[26] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person & Sarfraz Khurshid (2014): *Feedback-driven Dynamic Invariant Discovery*. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, ACM, New York, NY, USA, pp. 362–372, doi:10.1145/2610384.2610389.